ALMANAC

RELIABLE SMART SECURE
INTERNET OF THINGS FOR SMART CITIES

(FP7 609081)

ID 5.4 Ontologies and Semantic Representation Layer Prototype 1

Date 2014-11-30 – Version 1.0

Published by the Almanac Consortium

Dissemination Level: Restricted

Project co-funded by the European Commission within the 7th Framework Programme
Objective ICT-2013.1.4: A reliable, smart and secure Internet of Things for Smart Cities

Document control page

Document file: ID5.4.1 Ontologies and Semantic Representation Layer Prototype.docx
Document version: 0.4
Document owner: Jaroslav Pullmann (Fraunhofer FIT)

Work package: WP5 – Abstraction of Smart City Resources
Task: T5.3 Ontologies and Semantic Representation
Deliverable type: P

Document status: ☒ approved by the document owner for internal review
☒ approved for submission to the EC

Document history:

Version	Author(s)	Date	Summary of changes made
0.1	Jaroslav Pullmann (Fraunhofer FIT)	2014-11-26	First version with contributions from ISMB.
0.2	Jaroslav Pullmann	2014-12-01	Updated and extended version according to reviewer's feedback
0.3	Jaroslav Pullmann	2014-12-05	Updated according to new reviewer's feedback.
1.0	Jaroslav Pullmann	2014-12-05	Final version.

Internal review history:

Reviewed by	Date	Summary of comments
Dario Bonino (ISMB)	2014-12-05	Approved with minor comments
Matts Ahlsen (CNET)	2014-12-02	Approved with comments

Index:

1. Executive summary

This internal deliverable reports the design and implementation of the first prototype of the Semantic Representation Layer (further abbreviated as SRL) and data models provided in M15.

The work was undertaken within T5.3. The preliminary design ideas and the planned work timeline defined in the description of this task are first summarized. Then, according the twofold purpose of this task, the development of a Semantic Representation Layer (software) and the creation of a Smart City ontology (data models), is discussed. The document concludes with an outlook on future developments, up to the M30 milestone, reached by the final iteration of this deliverable, namely the ID5.3.2.

2. Introduction

2.1 Purpose, context and scope of this deliverable

This deliverable supplements the source code documentation of the prototype (see section 11) by providing background information on the conceptual analysis, the architecture design and the implementation choices taken as part of the overall efforts involving the so-called Semantic Representation Layer (SRL). It further aims at providing an informal specification and, a reference, for:

- Using and configuring the SRL and its subcomponents
- Interfacing the APIs and services exposed by SRL
- Understanding the data formats (media types) supported by SRL
- Defining the data modelling principles to be followed in the ALMANAC's Smart City Ontology
- Cataloguing 3rd-party vocabularies to be reused in the ALMANAC's Smart City Ontology

The final deliverable ID5.4.2 will complement this document in order to serve as the main information source for SRL and Smart City ontology usage and development.

2.2 Task goals according to DOW

The following sections summarize relevant statements, informal requirements and expected outcomes laid down in the description of the respective task T5.3 within project's Description of Work (DOW). They will serve as a reference for further formalization (requirements, specification) and inform the architecture design. Individual DOW statements are italicized and followed by a short discussion.

2.3 Semantic Representation Layer (SRL)

First category of DOW statements covers the software to be developed within T5.3 with implications on the overall ALMANAC platform:

1. *Distributed knowledge management infrastructure*: Leveraging current Linked Data technologies and vocabularies "distribution" is at heart of SRL's graph-based data model. The SPARQL 1.1 Standard is internally applied for data processing. It natively supports the fusion of distributed sources via query federation[1].
2. *Tools to systematically maintain, extend and share knowledge stored in the Smart City Ontology*: SRL applies and extends, whenever needed, the reference standards (SPARQL Graph protocols, see Annex I – Overview of SPARQL 1.1. Protocols) to implement a novel concept of RESTful manipulation of semantic resources.
3. *(Web) APIs to support generation and consumption of linked open data*: SRL offers remote and local (Java-based) service interfaces for easy integration with client and server components.
4. *Integration with semantic-aware applications and other ALMANAC components (Virtualization Layer)*: Close integration with ALMANAC and external components will be pursued. Resources exposed by the Smart City Resources Adaptation Layer (SCRAL) will be described by exploiting vocabularies and metadata defined at the SRL level, and low-level functions will be "represented" through a uniform modelling paradigm enabling better separation between upper layers of the architecture and low-level issues and paradigms. The Virtualization layer will support the core SRL services by offering a general virtualization framework in which SRL services will be complemented by advanced routing and transformation services, as an example.

[1] http://www.w3.org/TR/sparql11-federated-query/

2.3.1 Smart City Ontology

The following set of DOW statements lists ideas and expectations on semantic models to be provided by the task T5.3:

1. *Commonly shared semantic models and basic vocabularies*: Standard and commonly used models will be consulted, documented and used where appropriate. An authoritative catalogue of modelling principles and guidelines will be established.
2. *Semantically enriched description of offered services*: The task will investigate the use cases and required coverage of semantic service descriptions (e.g. service lookup, invocation etc.). It will rely on extended research activities like OWL-S[2], WSMO[3] and Linked USDL[4].
3. *Semantically enriched data streams*: Semantic event models or annotation of events will make transient streaming data accessible to continuous semantic queries[5]. The task will define requirements for semantic event modelling, processing and integration with the Data Fusion activities in WP6.
4. *Data quality annotation in Smart City applications*, i.e. quality attributes for the annotation of measurements and for semantic resolution of values against domain ontologies (support for WP6): This expectation will partially be covered by semantic event annotations (item 3).
5. *Adopt open standards to manage any relevant physical or logical attribute* (e.g. geographical position, time information, physical measurements, etc.): The task will strive for an extensive coverage of a semantic domain description.
6. *Models both service- and event- oriented aspects*, delivering a seamlessly extensible semantic model.
7. *Ensures semantic interoperability between components*: Semantic interoperability may, at least partially be reached, by translation between explicitly modelled interfaces. This requirement is already covered by item 2)
8. *Supports navigation among Smart City concepts/resources*: Search and navigation / exploration are among the core functionalities to be provided by the SRL. The supported paradigms and implementation in context of ALMANAC end user interfaces have to be investigated yet.
9. *Links to Virtualization Layer and the ALMANAC model via annotation of URI-identified resources*: Unified Resource Identifiers are the building blocks of the (Semantic) Web and they are used throughout the ALMANAC platform to identify any kind of internal or external resources.
10. *Supports the Virtual Entity concept of IoT-A, semantically represents Virtual Entities and their mapping to Physical Entities*: IoT-A[6] and ETSI-M2M[7] family of standards are considered crucial and they will be consulted in semantic modelling of networked infrastructures.
11. *Inference and querying features easing the management of the overall Smart City infrastructure*: Generic querying and inference capabilities of the software will be tailored to support concrete use cases by means of custom, domain specific, persistent queries.

2.3.2 Semantic Network Management

1. *Defines a common model of network capabilities*: This requirement has already been tackled with regards to measurement capabilities and, conditions of sensor nodes, by research activities like

[2] http://www.w3.org/Submission/OWL-S/
[3] http://www.wsmo.org/
[4] http://www.linked-usdl.org/
[5] https://www.insight-centre.org/content/semantic-discovery-and-integration-urban-data-streams
[6] http://www.iot-a.eu/public
[7] http://www.etsi.org/technologies-clusters/technologies/m2m

Semantic Sensor Networks[8], or the Sensor Planning Service[9]. Such an existing state of the art will be exploited as starting point, and the task activities will further integrate existing models by investigating approaches to represent communication and QoS capabilities of various network types.

2. *Coherently represent resources, capabilities and performance of the different heterogeneous networks*
3. Describes the diverse (network) management domains, the heterogeneous networks managed by different administrative domains

2.4 Deliverables

The following list relates this deliverable to the other ongoing and future project activities:

- ~~D5.1.1 Design of the abstraction framework and models (ISMB, M12/Aug14)~~
- D5.1.2 Design of the abstraction framework and models (FIT, M24/Aug15)

  The deliverables D5.1.1 and D5.1.2 present the overall approach to abstraction, virtualisation and semantic modelling of Smart City resources. They focus on interaction and integration of the respective prototype components Adaptation Layer (T5.1), Virtualization Layer (T5.2) and Semantic Representation Layer (T5.3).

- **ID5.4.1 Ontologies and Semantic Representation Layer Prototype (FIT, M15/Nov14)**
- ID5.4.2 Ontologies and Semantic Representation Layer Prototype (FIT, M30/Feb15)

  This focused deliverable and its successor cover the Semantic Representation Layer. They complement the deliverables ID5.2 Adaptation Layer Prototype and ID5.3 Virtualization Layer Prototype.

[8] http://www.w3.org/2005/Incubator/ssn/ssnx/ssn#MeasurementCapability
[9] http://www.opengeospatial.org/standards/sps

3. Semantic Representation Layer (SRL)

3.1 Overview

The main purpose of this service layer is to maintain, and provide, an easy programmatic access to rich, graph-based, domain models developed as part of the T5.3 activities. Its application scope is completely separate from the one of the "Storage manager", as the main goal of the SRL is to store, and make accessible, reference models and notations and to offer means to perform query and inference on such data. The SRL it is not intended to provide mass storage of measurement data, events, or the like, as reported in Table 1.

Table 1. Role of the Storage Manager in comparison with the Semantic Representation Layer

Storage Manager	Semantic Representation Layer
Tables, prescriptive relational schema, static[10]	Graphs, descriptive schema, dynamic
Large amount of flat data (recent and historical values)	Complex metadata, recent values
Explicit value selection (SQL)	Graph pattern matching (SPARQL)

Implementation of this component is largely based on the open source LinkSmart Metadata Framework (LSM). The ongoing development of the LSM framework is part of the T5.3 activities. SRL will provide project specific-extensions, customization and integration of LSM on behalf of ALMANAC.

3.2 LinkSmart Metadata Framework (LSM)

Recurrent and overlapping requirements among research projects led to the inception of the LinkSmart Metadata Framework, a set of utility services and data models to support working with RDF-based (meta)-data.

LinkSmart[11] for Java is an open-source middleware for P2P communication, mainly developed by Fraunhofer FIT and partners of the LinkSmart Association[12]. The Metadata Framework is a standalone extension module. While the framework is intentionally generic, it should provide an easy-to-use and complete infrastructure to enable individual projects and customers to implement domain-specific extensions on this base.

3.2.1 Design principles

The key principles followed by designing the LinkSmart Metadata Framework are summarized below.

Standards-compliant

The risk of creating a proprietary solution should be minimized by compliance to (Semantic Web) standards[13]. Related knowledge, support and open source software tools are re-used where possible.

Domain-agnostic

In order to increase re-usability and prevent the provision of custom APIs for every particular domain the APIs should be designed as generic and domain agnostic as possible, while allowing for definition, maintenance and querying of arbitrarily complex models.

Service-oriented

Service oriented architectures (SOA) focus in contrast to the object-oriented paradigm on functional interfaces/services exchanging merely "passive" data structures. Data management and retrieval has

[10] This is not necessarily true, the StorageManager acts as an interface to a number of different storage technologies. See D6.1 for reference.
[11] https://linksmart.eu/
[12] LinkSmart Association members are, among others, ISMB and CNet
[13] http://www.w3.org/standards/semanticweb/

been moved out of heavy-weight objects into a manageable set of services. SOA naturally fits various system architectures and protocols (Java/OSGi, HTTP/REST etc.).

Representation-based

In compliance to the service-oriented architecture, data structures exchanged by the LSM are deliberately simple and, optimized for serialization in widely adopted textual formats like XML, Turtle etc.

Resource-driven

Data manipulation and retrieval is implemented by adopting a RESTful paradigm, where resource representations are created, read, updated and deleted via a small set of uniform service interfaces.

3.2.2 Link Smart Metadata Architecture (LSM) outline

The Metadata framework maintains a graph representation of domain entities, together with their attributes, and of relationships between them, and offers means for querying and navigating such data. The framework acts as a transparent proxy to any triple-store compatible with the SPARQL 1.1 protocol[14]. It considerably augments the typical repository functions and interface coverage, by providing native Java/OSGi integration and remote HTTP APIs. Any client-related data (configurations, queries etc.) is persisted within the mediated triple store itself, allowing for their re-use and introspection by 3rd party tools. Thus valuable data assets and processing logic are made accessible to external applications preventing a vendor lock. The following sections will introduce the main constituents of the LSM architecture in reference to Figure 1.

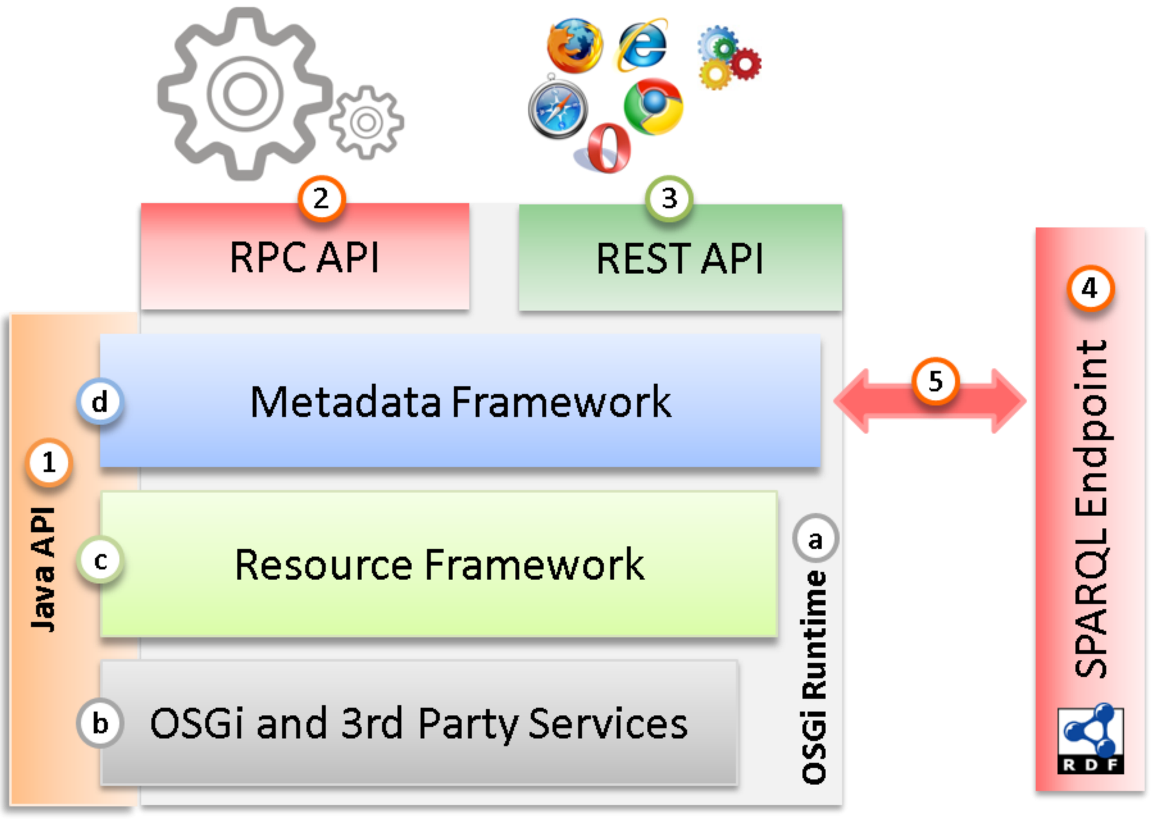


Figure 1. LinkSmart Metadata Framework outline

[14] Examples of such graph stores are Apache Fuseki or the Openlink Virtuoso Server.

OSGi Runtime environment (a)

A runtime environment implementing the *Open Services Gateway initiative* (OSGi) specification[15] for dynamic, service-oriented component systems in Java is the execution platform for LSM and the core building block. Its *Service Registry* acts as the central integration hub for publication and resolution of services.

OSGi and 3rd party services (b)

Next to the core OSGi services LSM leverages the *Service Component Runtime* (Declarative Service Specification[16]) for declarative component resolution and lifecycle management, the *Configuration-Admin* service[17] for management of persistent component configuration and other 3rd party services.

Resource Framework (c)

The *LinkSmart Resource Framework* defines a thin generic service layer for RESTful management of resources, including among others their life-cycle management, querying, conversion and validation.

Metadata Framework (d)

The *LinkSmart Metadata Framework* implements the resource management API for graph-based "semantic resources". It is compliant to and extends the SPARQL 1.1 web protocols (see section 6 for documentation) by provision of persistent SPARQL queries/updates and support of RESTful manipulation of resources at more fine grained levels (discrete graph fragments and individual properties) as explained in the next section 3.3.

Java data models and service interfaces (1)

According to the OSGi specification, data models and service interfaces of the installed bundles have to be explicitly exported and conversely imported via the manifest headers mechanism in order to be accessible to other bundles, allowing the framework to manage dependencies among them. The packages `eu.linksmart.resource` and `eu.linksmart.metadata` and their sub-packages comprise such a public interface of the abovementioned components (c) and (d).

Remote-procedure call (RPC) interfaces (2)

The RPC API serves the remote invocation of operations with custom semantics (e.g. data queries, actuation commands or calculations). These lower-level calls are often performed by higher levels services and not by human clients, as indicated by the icons. This interface handles SPARQL 1.1. Protocol requests and the invocation of custom persistent queries and updates.

Representational state transfer (REST) interfaces (3)

The REST API serves the retrieval and manipulation of resources identified by a Universal Resource Identifier (URI) by exchange of resource representations, i.e. their discrete textual serializations. The requests are issued towards a uniform service interface built upon the standard HTTP methods (verbs), mainly `GET`, `PUT`, `POST` and `DELETE` and result in CRUD[18] operations on the internally maintained resource state. Different root path prefixes are used to distinguish the **RPC** and **REST** endpoint semantics:

`/service/...`	Root path for invocation of RPC-services (via overloaded `POST`).
`/resource/...`	Root path of RESTful operations, a uniform interface applied to any resource type. The Metadata REST API for handling of semantic resources maps to the path prefix `/resource/`**`semantic`**`/*`.

[15] http://www.osgi.org/
[16] http://wiki.osgi.org/wiki/Declarative_Services
[17] http://www.osgi.org/javadoc/r4v42/org/osgi/service/cm/ConfigurationAdmin.html
[18] http://en.wikipedia.org/wiki/Create,_read,_update_and_delete

The lifecycle of custom persistent queries and updates is managed via the REST interface, while their effective invocation takes place via the RPC API. See section 3.3.3 for detailed explanation of both remote interface types.

SPARQL 1.1. Endpoint (4)

An SPARQL endpoint is the address/reference to an HTTP listener capable of handling SPARQL Protocol requests (standardized interface to an RDF graph store).

SPARQL 1.1. Protocol (5)

SPARQL 1.1 Protocol[19] defines a convention for transmission and execution of SPARQL Queries and Updates to a SPARQL endpoint via HTTP. LSM uses this protocol in communication to the underlying RDF graph store.

3.3 Semantic resource handling

In contrast to file-based, document-oriented or RDBMS systems which share an implicit concept of entity boundaries (like a file, XML root element, JSON object or SQL table etc.) there is no simple mean, in graph-based RDF models, to shape boundaries of a particular sub-graph (entity description). The available options are:

1. Usage of **named graphs per entity** to provide context to and consolidate all statements about a single entity. The drawback of this approach is a missing support within the RDF abstract model itself (only available via TriG[20] serialization and manipulation level via SPARQL named graph support[21]), high fragmentation of the data base and difficulty to link and query.
2. Usage of **intermediate** blank nodes[22] to express boundaries, context, provenance etc. Drawback: this solution would require a proprietary data schema and framework implementing such a blank node traversal and would be incompatible with most of the existing vocabularies.
3. Usage of **SPARQL 1.1. Query and Update** languages to purposefully construct and manipulate entity sub-graphs. Drawback: development of a management framework and an initial configuration effort setting up the required queries/updates. Such a programmatic handling of graph fragments may optionally build on top of an additional named graph organization 1).

The latter solution 3) was selected for implementation in LSM, since it employs a standard tool chain and is not limited to a particular data schema.

3.3.1 Processing pipeline

As mentioned previously, SPARQL 1.1 query and update expressions are used to ad-hoc “construct” discrete entities out of the graph continuum. For this purpose the Metadata Framework adopts the generic resource processing pipeline of the LinkSmart Resource Framework as depicted inFigure 2. Annex section 10 provides a commented example of the overall request chain.

[19] http://www.w3.org/TR/sparql11-protocol/
[20] http://www.w3.org/TR/trig/
[21] http://www.w3.org/TR/sparql11-query/#namedGraphs
[22] http://en.wikipedia.org/wiki/Blank_node

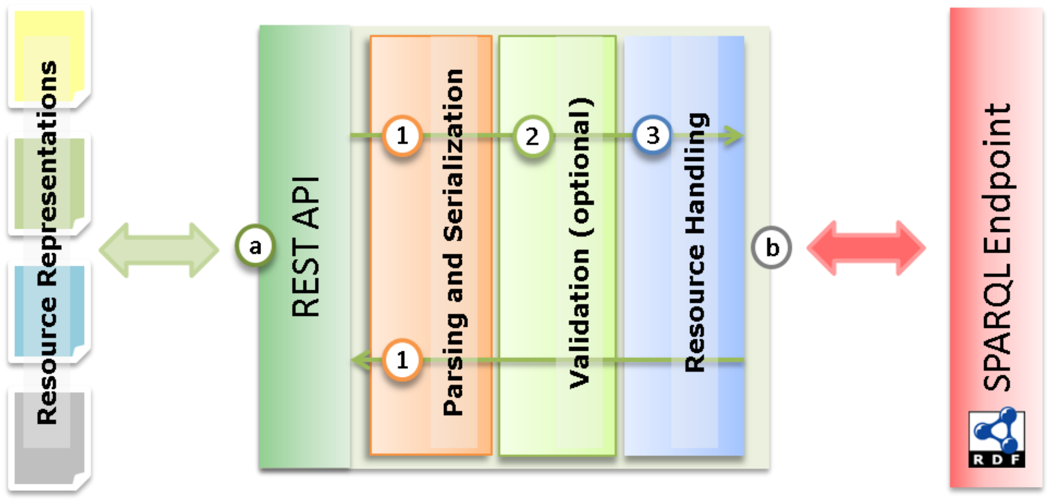


Figure 2. Semantic resource processing pipeline

REST-API Clients (a)

External clients of the REST API interact by invoking the uniform interface and exchanging resource representations. These vary in terms of resource type and media type (serialization).

Parsing and serialization handlers (1)

The textual resource representations are parsed on input or serialized on output to/from an internal graph model (Apache Jena `Model`[23]). Independently of the resource type a `ResourceRequest` object wrapping the resource is created in accordance to the request properties (HTTP headers, query or form parameters) and passed to the pipeline.

Input validation handlers (2)

Resources passed along the `PUT` or `POST` requests are optionally validated by a `ResourceValidator` service. Validation of semantic resources is by default implemented via a SPARQL `ASK` query. It returns true, for a valid (matching) graph input, false otherwise.

Resource handlers (3)

Based on the parameters of the `ResourceRequest` the most appropriate `RequestHandler` service is chosen to handle the request. For this purpose `RequestHandler` instances are expected to indicate their applicability (type of request they are capable of handling and identifiers of individual resources or resource types they apply to). The Metadata Framework ships with a set of default request handlers that serve operations on simple semantic resources out of the box. Such, for example, if no custom handler could be found the default `ReadHandler` implemented via a SPARQL `CONSTRUCT` query is used.

SPARQL 1.1 endpoint request (b)

A standard SPARQL 1.1 Protocol request is created and issued against the remote RDF graph store.

3.3.2 Provisioning of semantic resource handlers

While an application bundle may choose to manually register `ResourceHandler` services (e.g. implemented in Java), the Metadata Framework automatically synchronizes with the underlying SPARQL endpoint and turns any `ResourceHandler` description found into an equivalent OSGi service as depicted in Figure 3.

[23] http://jena.apache.org/documentation/javadoc/jena/com/hp/hpl/jena/rdf/model/Model.html

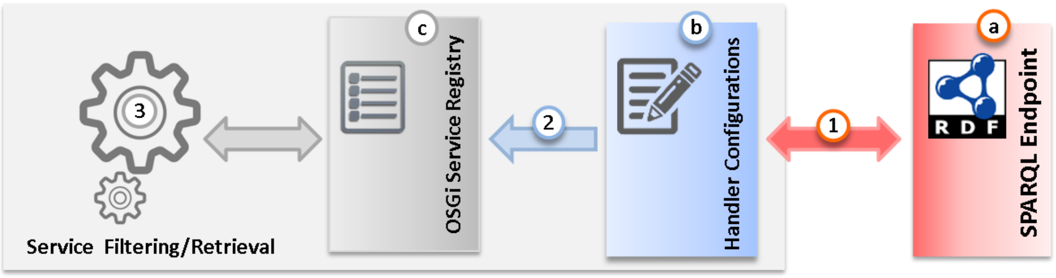


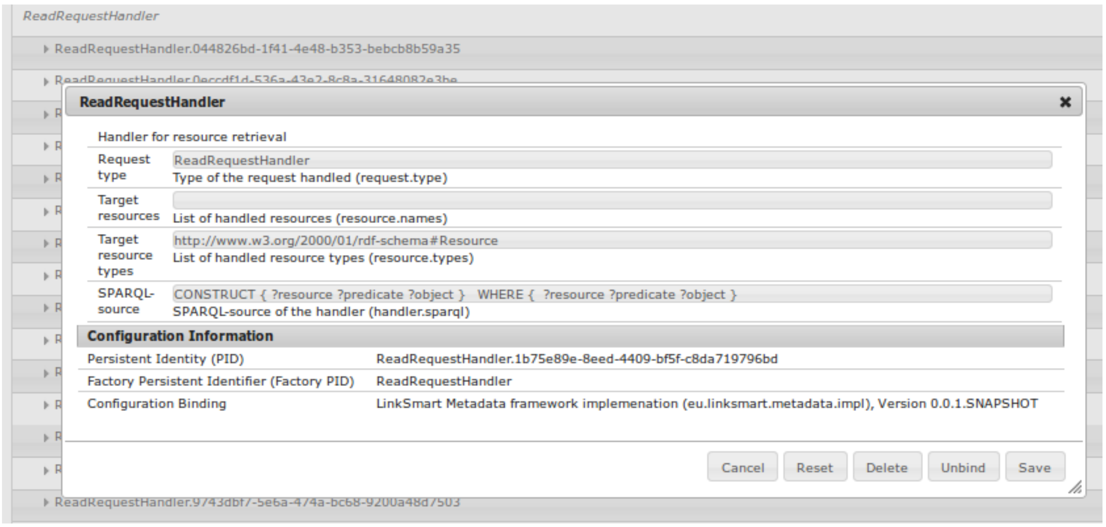
Figure 3. Synchronization of the semantic resource handlers and configuration

SPARQL 1.1 endpoint (a)

The underlying triple store is searched for `ResourceHandler` definitions. These are retrieved via SPARQL 1.1. Protocol query invocations **(1)**.

Handler configurations (b)

The Metadata Framework instruments the standard OSGi `ConfigurationAdmin` service to create (or update existing) `Configuration`[24] object for every retrieved handler definition. These configurations are explicit, OSGi-side representation of the handler logic accessible to manual editing, deletion etc. via an administrative user interface (e.g., the Apache Felix WebConsole[25]) as shown in Figure 4:


ReadRequestHandler

ReadRequestHandler.044826bd-1f41-4e48-b353-bebcb8b59a35

ReadRequestHandler.0eccdf1d-536a-43e2-8c8a-31648082e3be

ReadRequestHandler

Handler for resource retrieval

Request type	ReadRequestHandler Type of the request handled (request.type)
Target resources	 List of handled resources (resource.names)
Target resource types	http://www.w3.org/2000/01/rdf-schema#Resource List of handled resource types (resource.types)
SPARQL-source	CONSTRUCT { ?resource ?predicate ?object } WHERE { ?resource ?predicate ?object } SPARQL-source of the handler (handler.sparql)

Configuration Information

Persistent Identity (PID)	ReadRequestHandler.1b75e89e-8eed-4409-bf5f-c8da719796bd
Factory Persistent Identifier (Factory PID)	ReadRequestHandler
Configuration Binding	LinkSmart Metadata framework implemenation (eu.linksmart.metadata.impl), Version 0.0.1.SNAPSHOT

ReadRequestHandler.9743dbf7-5e6a-474a-bc68-9200a48d7503

Figure 4. Screenshot of a resource handler configuration

Service registry (c)

The *Service Component Runtime* (implementation of the OSGi *Declarative Services* Specification[26]) monitors the available configurations and instantiates and registers or, eventually, discards corresponding `ResourceHandler` instances (SCR components) within the OSGi Service registry. The Metadata Framework may resolve and obtain those services (3) by filtering the OSGi Service registry.

[24] http://www.osgi.org/javadoc/r4v42/org/osgi/service/cm/Configuration.html
[25] http://felix.apache.org/site/apache-felix-web-console.html
[26] http://wiki.osgi.org/wiki/Declarative_Services

3.3.3 Remote HTTP APIs

The remote HTTP service interfaces (on top of Figure 1) are the main entry point for LSM web clients. They implement and extend SPARQL 1.1 Web Protocols allowing for RPC (Remote Procedure Call) and REST-style interactions:

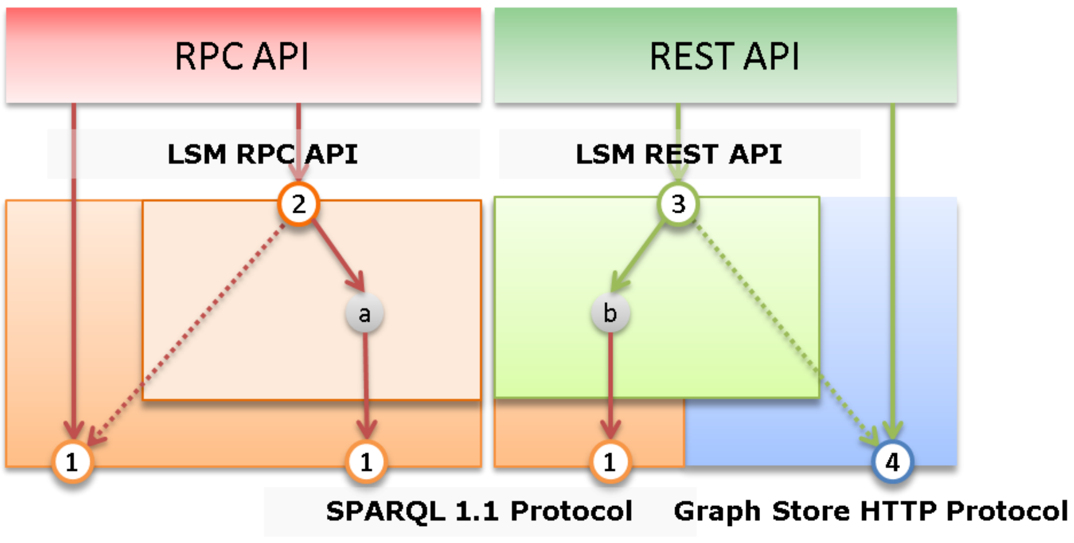


Figure 5. LSM Remote APIs

The RPC API comprises the *invocation of services* to query or update data whereas the REST API serves *the retrieval and manipulation of (textual) resource representations*. Client requests to remote HTTP interfaces offered by the LinkSmart Metadata Framework are transparently proxied and transformed to invocation of standardized interfaces on a SPARQL endpoint **(1)** and **(4)** at the bottom.

SPARQL 1.1 Protocol[27] **(1)**

Standard interface exposed by a SPARQL endpoint, **mandatory** for (2).

LinkSmart Metadata Framework RPC API (2)

The RPC API accepts query or update invocations. Standard requests are immediately proxied to the endpoint (1), custom requests are internally processed (a) and forwarded to endpoint (1). "Processing" may involve query rewriting, resolution and parameterization of stored queries and further request enhancements etc.

LinkSmart Metadata Framework REST API (3)

If available, the REST API will proxy standard Graph protocol requests to (4), otherwise standard requests are converted into invocation of the RPC API (1) according to predefined translation rules. The API extends standard API (4) by exposing wide range of resources (URIs) for sub-graph manipulations (b).

Graph Store HTTP Protocol[28] **(4)**

Graph Store protocol is an **optional** standard interface for RESTful operations on graph level.

3.3.4 SPARQL Protocol Service URL Design

The SPARQL Protocol standard does not predefine a URL convention for SPARQL Protocol Services (endpoints). Naming of the query/update or graph protocol endpoints is not considered. It only states

[27] http://www.w3.org/TR/sparql11-protocol/
[28] http://www.w3.org/TR/sparql11-http-rdf-update/

that the implementation of the data-modifying "update" operation shall be optional for security reasons:

```
SPARQL protocol services may remove, insert, and change underlying data via
the update operation. To protect against malicious or destructive updates,
implementations may choose not to implement the update operation. Alter-
natively, implementations may choose to use HTTP authentication mechanisms
or other implementation-defined mechanisms to prevent unauthorized invoca-
tions of the update operation.
```

Even the specification examples mention different endpoints (`/sparql` vs. `/test`, see sections 3.1 and 3.2). We chose to separate the service endpoints for *read-only* (query) and *read-write* (update) operation semantics in order to support different security policies, specifically tailored on the allowed interactions between externals service consumers and resources managed by the SRL (for the same reason the underlying OSGi services are distinguished). The SPARQL Protocol Services are assumed to co-exist in a context of other RPC services provided by the platform, therefore the standard RPC prefix path "`/service`" is assumed:

Table 2. RPC-endpoints for SPARQL Protocol in LinkSmart Metadata Framework

Endpoint URL	Description
`/service/sparql/query`	RPC-endpoint for execution of ad-hoc and persistent SPARQL 1.1 queries (read-only semantics, no data updates)
`/service/sparql/update`	RPC-endpoint for execution of ad-hoc and persistent SPARQL 1.1. updates (write-only semantics, no data retrieval)

3.3.5 Persistent queries and updates

As any other resource SPARQL query or update statements might be stored using the REST-API and invoked by a request to above mentioned endpoints:

Table 3. Examples of SPARQL RPC-endpoint invocations

Request example	Description

`POST /service/sparql/update/`**`res:id251`**
 	
`temperature=10.3`
	Request to update a value by executing a SPARQL Update specified inline by its compact resource URI. The required parameters are supplied as `application/x-www-form-urlencoded` request payload.

`POST /service/sparql/update`
 	
`update-name=res:id251``&temperature=10.3`
	Request to update a value by executing a SPARQL Update referenced by the mandatory parameter `update-name`.

3.3.6 SPARQL Graph Protocol URL Design

The standard Graph Store HTTP Protocol operates on level of entire graphs which are comparable to relational databases, NoSQL collections or file folders. Content manipulation at graph-level is too coarse grained for applications targeting large number of resources. The main contribution of the LinkSmart Metadata Framework is the definition and implementation of a standards-based infrastructure to allow the retrieval and modification of resources at configurable sub-graph levels. Please refer to annex 6.2 for commented examples of the protocol usage and to section 3.3.1. for details on the extension framework.

3.3.7 Summary of the REST API for RDF

A conceptual summary of the current REST API is given. All requests URLs start with the path `<prefix>`: `/resource/semantic`. See annex chapter 10 for commented examples.

Function	Request	Description

Resource listing	`GET /<prefix>/<class_CURIE>`	Request to list resource representations of given type (class). Offers pagination and offset (query) parameters via limit and offset parameters. Given the amount of generated data a more concise formatting might be used than for "Resource retrieval".
Resource creation	`POST /<prefix>/<class_CURIE>` `...representation...`	Request to create a new resource of given type based on supplied representation. The resource URI will be generated and the resultant resource will be typed by this class.
Programmatic resource creation	`POST /service/sparql/update` `update-name=...&foo=...`	Request to create a new resource by executing an annotated SPARQL Update configured via query parameters. The re-source URI will be generated.
Resource retrieval	`GET /<prefix>/<resource_CURIE>`	Request to retrieve a representation of given resource in a particular representation (format).
Resource replacement	`PUT /<prefix>/<resource_CURIE>` `...representation...`	Request to replace an existing resource by supplied representation.
Resource extension	`POST /resource/<resource_CURIE>` `...partial representation...`	Request to augment the given resource by given **partial** representation. The supplied graph will be unified with contents of the repository.
Resource deletion	`DELETE /resource/<resource_CURIE>`	Request to delete specified resource.

3.3.8 Storage context

The storage context of a semantic resource is defined by the *repository* URL (comparable to a relational database system) and its *graph* (comparable to a database or collection), which is either the implicit, "default" graph or a "named graph" identified by a URI. The notion of a named graph[29] as management concept was introduced by the SPARQL standard[30], in which queries operate on RDF Datasets[31] – the union of a default graph and selected named graphs. In compliance with the Graph Store HTTP Protocol, query parameters `default`, `graph` and `repository` are used to express resource's storage context:

Storage context supplied as (optional) query parameter	
`/resource/...`	Implicit reference to the default graph and a pre-configured, default repository
`/resource/...?default`	Explicit reference to the default graph and repository
`/resource/...?repository=<repository>`	Explicit reference to a non-default repository URL
`/resource/...?repository=<repository>&graph=<graph>`	Explicit reference to a non-default repository and graph URI. Default configuration is used if both are omitted

[29] http://en.wikipedia.org/wiki/Named_graph
[30] http://www.w3.org/TR/sparql11-query/#namedGraphs
[31] http://www.w3.org/TR/sparql11-query/#rdfDataset

4. Smart City Ontology

4.1 Design principles and guidelines

This section tracks down design assumptions, and guidelines to better support shared and explicit understanding of principles lying at the basis of the Smart City ontology design. Currently the following design guidelines have been identified:

Vocabularies

- Prefer established vocabularies vs proprietary, "home-made" schemes.
- More effort on (re)search and selection than on development of an own schema, which must always be motivated (e.g., no other schema available/applicable for the given knowledge domain).
- Consider aspects like: maturity, expressiveness, community and software support.
- Use possibly one, single vocabulary for any particular purpose, avoiding redundant descriptions.

Annotation, mapping

- Provide mappings to existing established vocabularies whenever possible
- If applicable, prefer built-in annotation properties[32] of the W3C Semantic Web standards: (`rdfs:label, rdfs:comment` etc.)

Technical choices

- Use typed blank nodes (bnodes) wherever possible. Since bnodes do not have an external identity assigning a meaningful type will increase their usability with regards to resolution and manipulation.

4.1.1 Relevant vocabularies

As a general guideline, standardized and relevant models should be re-used whenever possible. This guideline is full-filled iteratively and continuously during the task activities and starts from a preliminary survey of currently adopted modelling frameworks. Among many models and ontologies defined in the Linked Open Data community, a subset of relevant resources has been identified and includes others:

- The Semantic Sensor Network Ontology (SSN);
- The DogOnt Ontology Modeling for Intelligent Domotic Environments;
- The SCRIBE IBM Smart City ontologies (currently not published yet);
- The models listed in the Smart Cities ontology catalogue, among others: Places Ontology, Cadastre and Land Administration Thesaurus (CaLAThe).
- The models listed in the Linked Open Vocabularies (LOV) vocabulary repository;
- The Places ontology, a light-weight ontology for describing places of geographical interest;
- The Schema.org vocabulary for representing well known geographical properties, e.g., latitude and longitude;
- The GoodRelations ontology for representing any kind of goods, products and services, and the relations involving their offering, exchanges, etc.;
- The MUO unit of measure ontology, representing unit of measures according to the UCUM vocabulary (including International System of Measures values and other relevant units);
- The vcard ontology for representing name and addresses of people and organizations;

[32] http://www.w3.org/TR/owl2-syntax/#Annotation_Properties

- The GeoSPARQL vocabulary and functions, to support representation and querying of geo-spatial data.
- The GeoNames ontology for representing cities and other geographical entities.

Beyond immediate "SmartCity" aspects (administration, environment, utility services etc.) the model must also tackle issues related to network infrastructure in order to implement the foreseen "Semantic Network Management". Normally only fragments of the imported 3rd party schemas are needed. See Annex III – Reference of common schema constructs for an overview of the most relevant constructs, their function along with usage example and justification.

4.1.2 Ontology Development

A mature web-based ontology editor was chosen to support distributed, collaborative development and annotation of the ontology files: WebProtege[33]. The editor instance with pre-loaded ontologies, including the current ALMANAC Smart City ontology for waste management provided by ISMB is available online at: **`http://almanac.fit.fraunhofer.de:8080/webprotege`**

4.2 WasteBin ontology

A first, preliminary, modelling effort for representing smart city data with respect to waste management issues has been performed by ISMB, leading to the definition of a light-weight "waste bin" ontology (see annex section 9 for ontology source in Turtle format).

The ontology rationale is to focus on waste-related issues with an open, extensible and scalable approach. According to this design goal, no assumption is performed on the remaining smart city data, and modelling is restricted to the aspects specifically related to waste collection and management, e.g., by representing waste bins, type of disposables generated by the citizenship and so on. All represented concepts leverage existing, well known definitions of properties (and classes / super-classes) with a typical Linked Open Data approach. In such a sense, for example, the city concept is directly linked (`owl:equivalentClass`) to the corresponding schema and places concepts.

The ontology mainly represents waste bins, waste types and the geographical / administrative context in which they are deployed. It is organized along three main hierarchies (*isA* or *partOf*) respectively rooted at the classes: `WasteBin`, `City` and `Waste`.

The former represents different bin types such as bins for gathering organic waste, glass, paper, etc. Six different types of bins are represented including: dry waste, glass and aluminum, organic, paper, plastic and used clothes bins. These types are clearly emerging from a national (Italian, as one of the ALMANAC end users is the Turin's municipality) "standpoint" on waste collection, but it can easily be extended to represent typical waste collection in Europe.

The second hierarchy of objects, is organized along a *partOf* containment tree and includes both physical (`City`) and administrative (`District` and `Quarters`) concepts. Since the main concepts represented in this tree are widely used in several knowledge domains, and application scenarios, the WasteBin ontology definition is mainly built through equivalence classes, and it only adds geo-spatial information for automatically inferring spatial containment between waste bins and administrative / physical regions.

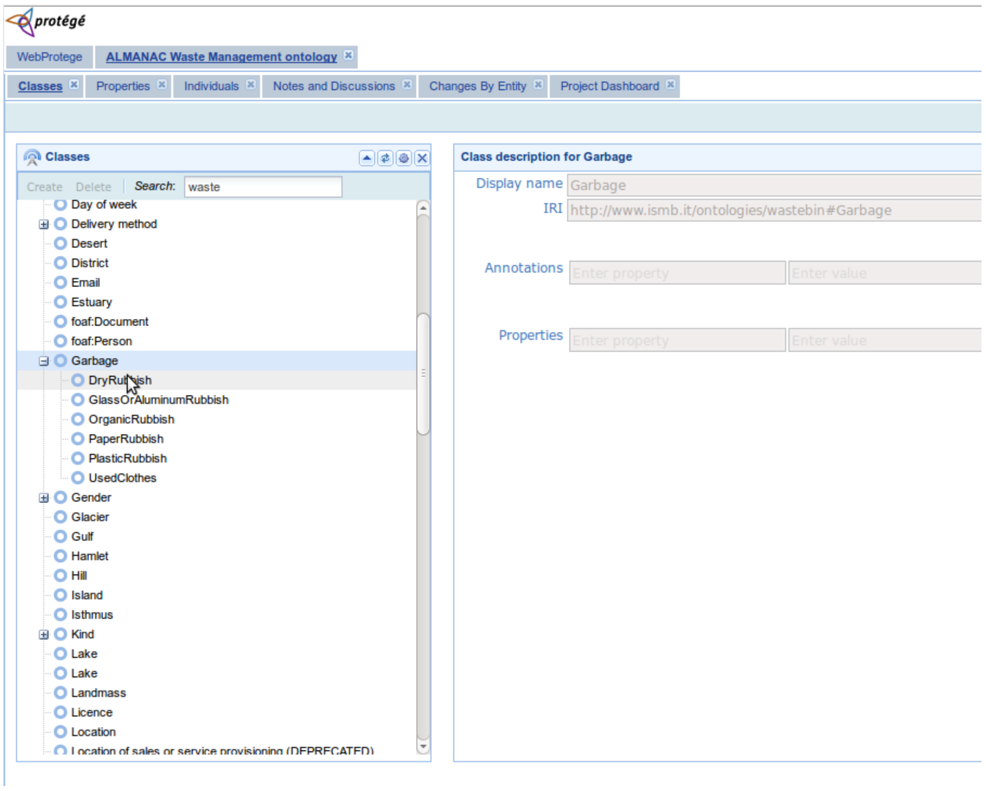
Finally, Waste types are defined to represent the kind of "garbage" collected by a specific bin and to represent, by means of suitable relationships, the waste generation behaviour of quarters and districts.

Under a more "technical" standpoint, the ontology counts 20 concepts, 6 object properties and 3 data-type properties; it features a SHIQ(D) DL expressivity and it is interconnected with 6 different and widely recognized vocabularies including: the geonames ontology, the places ontology, the GeoSPARQL vocabulary, the Schema.org vocabulary, the VCard and Good Relations and the Unit of Measurment ontologies (MUO). A full instantiation representing the public information available on

[33] https://github.com/protegeproject/webprotege/releases

waste bins deployed in Turin is provided as initial use case, and models around 29k waste bins, one city, 10 districts and 25 quarters.

A demo application with almost 28.000 generated instances of `WasteBin` types mapped to different locations in city of Turin was successfully demonstrated at the IoT360 Conference[34] in Rome.



Search: waste

Display name: Garbage

IRI: http://www.ismb.it/ontologies/wastebin#Garbage

Figure 6. WasteBin ontology loaded in WebProtege

[34] http://iot-360.eu/agenda/

5. Conclusion and future work

This deliverable presented the design and initial implementation of the Semantic Representation Layer and accompanying ontologies. It covers the continuous development and improvement of the underlying LinkSmart Metadata Framework. The future work on SRL will focus on:

- Extension of the existing ontology by data models covering ALMANAC specific SmartCity resources (sensor and communication networks, devices, urban environment).
- Extension of the RESTful API allowing for exchange of semantic resource descriptions. Custom, project specific queries will be defined to provide synthetic view on the data and serve requirements of SRL clients.
- Integration of project internal and external components (SCRAL, Virtualization layer etc.).

6. Annex I – Overview of SPARQL 1.1. Protocols

This section provides an overview of the two standard SPARQL 1.1 protocols for performing remote read-write operations on RDF-graph content:

1. SPARQL 1.1 Protocol: HTTP-based protocol for performing supplied SPARQL 1.1. Query and Update requests
2. SPARQL 1.1 Graph Store HTTP Protocol: RESTful-protocol for performing CRUD operations on RDF content

6.1 SPARQL 1.1 Protocol

Based on HTTP this application-level protocol defines conventions for executing SPARQL Protocol operations ("query" or "update"). These are defined by a configuration of an HTTP method, query string parameters and a message payload. Generally speaking a request contains a query or update string and optionally indicates the RDF Dataset (in terms of graph URIs) to be applied on. An RDF Dataset comprises one graph, the "default" graph (does not have a name) and zero or more named graphs each identified by an URI. While the data set may be specified within the SPARQL string, the query parameters take precedence. Only the listed (named) graphs will be considered while matching the query.

6.1.1 Query operation

The query operation comprises invocation of a supplied SPARQL 1.1. Query string on a remote SPARQL endpoint. It is defined by:

- HTTP method: `GET`, `POST` (safe - no side effects, since read only)
- Query string parameters:
    - `query`: SPARQL 1.1 Query string
    - `default-graph-uri`: equivalent to FROM statement indicating an explicit default/background graph
    - `named-graph-uri`: equivalent to FROM NAMED statement indicating a named graph to match against via GRAPH keyword
- Literal message payload
- Request: SPARQL 1.1 Query string, if not specified by parameter
- Response: Result representation (a serialization of `ASK`, `SELECT` or `CONSTRUCT` query result), depends on content type supported by client

Their possible/permitted combinations are the following:

	HTTP Method	**Query String Parameters**	**Request Content Type**	**Request Message Body**
a) query via POST directly	POST	`default-graph-uri` (0 or more) `named-graph-uri` (0 or more)	`application/sparql-query`	Unencoded SPARQL query string
b) query via URL-encoded POST	POST	None	`application/x-www-form-urlencoded`	URL-encoded, ampersand-separated query parameters: query (exactly 1) default-graph-uri (0 or more) named-graph-uri (0 or more)

c) query via GET	GET	`query` (exactly 1) `default-graph-uri` (0 or more) `named-graph-uri` (0 or more)	`None`	None

6.1.2 Update operation

The update operation comprises the invocation of a supplied SPARQL 1.1. Update string on a remote SPARQL endpoint. It is defined by:

- HTTP method: `POST`
- Query string parameters:
  - `update`: SPARQL 1.1 Update string
  - `using-graph-uri`: equivalent to USING statement indicating an explicit default/background graph
  - `using-named-graph-uri`: equivalent to `USING NAMED / WITH` statement indicating a named graph to match against
- Literal message payload
  - Request: SPARQL 1.1 Update string, if not specified by parameter
  - Response: HTTP status code (no payload)

Their possible/permitted combinations are the following:

	HTTP Method	Query String Parameters	Request Content Type	Request Message Body
a) update via POST directly	POST	using-graph-uri (0 or more) using-named-graph-uri (0 or more)	`application/sparql-update`	Unencoded SPARQL update request string
b) update via URL-encoded POST	POST	None	`application/x-www-form-urlencoded`	URL-encoded, ampersand-separated query parameters. update (exactly 1) using-graph-uri (0 or more) using-named-graph-uri (0 or more)

Since the above request alternatives are purely syntactical variants chosen for convenience we propose stick to one single, consistent representation for internal implementation. The request variant a) HTTP POST with plain SPARQL Query/Update payload and optional query parameters (green lines) was selected for implementation in LSM to encode persistent queries.

6.2 SPARQL 1.1 Graph Store HTTP Protocol

This protocol targets retrieval and manipulation of entire RDF graphs by means of RESTful operations (i.e. exchange of resource representations and not the interpretation of query/update strings). The standard distinguishes scenarios where the target graph is either directly identified by the request URL (a) or indirectly by an URI passed as a query parameter (b and c).

Table 4. Graph identification in SPARQL Graph Store HTTP Protocol

Graph Store HTTP Protocol request example	Description
`GET http://example.com/rdf-graphs/employees`	a) Direct, resolvable graph reference via the request URI. Graph's identifier (URI)

	coincides with its storage / management location (URL). The graph content is manipulated by issuing REST requests directly to this URL. This approach follows the core principle of the WWW being a net-work of interlinked resources resolvable via HTTP requests. On the other side Resource URIs are bound to a concrete transport protocol and expose protocol. Changes to deployment architecture has to be kept transparent, otherwise the URIs will break.
GET http://example.com/rdf-graph-store?graph=**http%3A//www.example.com/other/graph**	b) Indirect reference to a named graph (absolute URI) is supplied via query para-meter - the graph URI is distinct from management endpoint's URL.
GET http://example.com/rdf-graph-store?default	c) Identifier of the implicit, default graph.

The standard defines operations on graph-level only (retrieval, replacement, deletion or augmenting a particular graph) and provides a mapping to equivalent SPARQL 1.1. Update operations:

Table 5. Overview of SPARQL Graph Store HTTP Protocol operations

Operation	HTTP Method	Request example	Equivalent SPARQL 1.1 Update (Implementation)
Retrieval of a named graph	`GET`	`GET /rdf-graph-store?graph=<graph_uri>`	`CONSTRUCT { ?s ?p ?o } WHERE { GRAPH <graph_uri> { ?s ?p ?o } }`
Retrieval of the default graph	`GET`	`GET /rdf-graph-store?default`	`CONSTRUCT { ?s ?p ?o } WHERE { ?s ?p ?o }`
Replacement of a named graph	`PUT`	`PUT /rdf-graph-store?graph=<graph_uri>`	`DROP SILENT GRAPH <graph_uri>; INSERT DATA { GRAPH <graph_uri> { RDF payload (Turtle) } }`
Replacement of the default graph	`PUT`	`PUT /rdf-graph-store?default`	`DROP SILENT DEFAULT; INSERT DATA { RDF payload (Turtle)}`
Deletion of a named graph	`DELETE`	`DELETE /rdf-graph-store?graph=<graph_uri>`	`DROP GRAPH <graph_uri>`
Deletion of the default graph	`DELETE`	`DELETE /rdf-graph-store?default`	`DROP DEFAULT`
Merge with/addition to a named graph	`POST`	`POST /rdf-graph-store?graph=<graph_uri>`	`INSERT DATA { GRAPH <graph_uri> { RDF payload (Turtle) } }`
Merge with/addition to the default graph	`POST`	`POST /rdf-graph-store?default`	`INSERT DATA { RDF payload (Turtle) }`

Operations and addressing issues on the sub-graph level of triples/resources are not covered. At this point the LSM semantic resource management provides a compliant extension of the standard.

7. Annex II – Overview of data formats and mime types

The following table enumerates mime types supported in interaction with the remote LinkSmart Metadata Framework APIs - request payload (`Content-Type` header) and response serialization (`Accept` header):

Content type	Usage	Description/Usage	File extension
`application/sparql-query`	IN	Content type of a literal SPARQL 1.1. query request	`.rq`
`application/sparql-update`	IN	Content type of a literal SPARQL 1.1. update request	`.ru`
`application/x-www-form-urlencoded`	IN	Content type of a parameter-encoded POST request (applies for queries and updates)	
`application/sparql-results+json`	OUT	SPARQL 1.1 Query Results JSON Format Simple JSON serialization of the result set using full URIs.	`.srj`
`application/sparql-results-compact+json`	OUT	Structurally equivalent serialization using compact URIs (CURIEs) encoded against a system-wide namespace mapping (introduced in LSM)	`.srcj`
`application/sparql-results+xml`	OUT	SPARQL Query Results XML Format	`.srx`
`text/csv`	OUT	SPARQL Query Results CSV Format (Comma Separated Values)	`.csv`
`text/tab-separated-values`	OUT	SPARQL Query Results TSV (Tab Separated Values)	`.tsv`
`application/rdf+xml`	IN/OUT	RDF/XML Syntax Specification (see Mime-type definition)	`.rdf`
`application/rdf+json`	IN/OUT	RDF 1.1 JSON Alternate Serialization (RDF/JSON)	`.rj`
`application/rdf-compact+json`	IN/OUT	Structurally equal serialization but using compact URIs (CURIEs) encoded against a system-wide namespace mapping (introduced in LSM).	`.rcj`
`text/turtle`	IN/OUT	RDF 1.1 Turtle (Terse RDF Triple Language)	`.ttl`

8. Annex III – Reference of common schema constructs

Topic / Function	Reference construct	Example	Description / Justification	Ontology / Namespace
Label / name	`rdfs:label`	`rdfs:label "Turin"`	Assigns a human-readable label to a given ontology class.	http://www.w3.org/2000/01/rdf-schema#
Comment	`rdfs:comment`	`rdfs:comment "The Turin Smart City"`	Provides a long comment or description to define a shared understanding of what a class represents.	http://www.w3.org/2000/01/rdf-schema#
Geometry	`geo:hasGeometry`	`geo:hasGeometry [geo:asWKT """Polygon((7.664075945991105 45.04861982156903,7.635273504256141 45.02273885185137,7.638375873441769 45.02188585137546,7.664075945991105 45.04861982156903))"""^^geo:wktLiteral] ;`	Describes the geometrical space (shape) taken by a given subclass of geo:Feature (object with geographic features such as latitute, longitude, shape, etc.)	http://www.opengis.net/ont/geosparql#
Feature	`geo:Feature`	`rdfs:subClassOf geo:Feature`	Describes a spatial object feature having a geographically referenced geometry	http://www.opengis.net/ont/geosparql#
Address	`vcard:hasAddress`	`vcard:hasAddress [vcard:street-address "Via VIGONE 62"; vcard:locality "Torino"] ;`	Describes the surface (mail) address associated to a given person / organization/ entity	http://www.w3.org/2006/vcard/ns#
Latitude	`s:latitude`	`s:geo [a s:GeoCoordinates;s:latitude 45.06798137; s:longitude 7.64859467]`	Describes the latitude of a (sub)class of s:Place, it may be used when a given instance has both a shape (Geometry) and a Point location.	http://schema.org#
Longitude	`s:longitude`	`s:geo [a s:GeoCoordinates;s:latitude 45.06798137; s:longitude 7.64859467]`	Describes the longitude of a (sub)class of s:Place, it may be used when a given instance has both a shape (Geometry) and a Point location.	http://schema.org#
Latitude - alternative	`wgs84:lat`	`vcard:hasGeo [a wgs84:Point; wgs84:lat`	Describes the latitude of a (sub)class of s:Place, it may be used when a given instance	http://www.w3.org/2003/01/geo/wgs84_pos#

		`45.06798137; wgs84:lon 7.64859467]`	has both a shape (Geometry) and a Point location.	
Longitude – alternative	`wgs84:lon`	`vcard:hasGeo [a wgs84:Point; wgs84:lat 45.06798137; wgs84:lon 7.64859467]`	Describes the longitude of a (sub)class of s:Place, it may be used when a given instance has both a shape (Geometry) and a Point location.	http://www.w3.org/2003/01/geo/wgs84_pos#

9. Annex IV – ALMANAC Waste bin ontology

```
@prefix s: <http://schema.org#> .
@prefix geo: <http://www.opengis.net/ont/geosparql#> .
@prefix geonames: <http://www.geonames.org/ontology#> .
@prefix vcard: <http://www.w3.org/2006/vcard/ns#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xml: <http://www.w3.org/XML/1998/namespace> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix wbin: <http://www.ismb.it/ontologies/wastebin#> .
@prefix places: <http://purl.org/ontology/places#> .
@prefix gr: <http://purl.org/goodrelations/v1#> .
@prefix muo: <http://purl.oclc.org/NET/muo/muo#> .


<http://www.ismb.it/ontologies/wastebin> rdf:type owl:Ontology ;
      owl:imports <http://schema.org/> ,
      <http://www.opengis.net/ont/geosparql> ,
      <http://www.w3.org/2006/vcard/ns> ,
      <http://purl.org/ontology/places> ,
      <http://purl.org/goodrelations/v1> ,
      <http://purl.oclc.org/NET/muo/muo> .




# object properties
wbin:yearlyWasteProduction rdf:type owl:ObjectProperty ;
      owl:ObjectPropertyDomain [a owl:Class; owl:ObjectUnionOf
(wbin:Quarter wbin:City wbin:District)] ;
      rdfs:domain [a owl:Class; owl:unionOf (wbin:Quarter wbin:City)] ;
      owl:ObjectPropertyRange wbin:YearlyWasteAmount ;
      rdfs:range wbin:MonthlyWasteAmount .

wbin:monthlyWasteProduction rdf:type owl:ObjectProperty ;
      owl:ObjectPropertyDomain [a owl:Class; owl:ObjectUnionOf
(wbin:Quarter wbin:City wbin:District)] ;
      rdfs:domain [a owl:Class; owl:unionOf (wbin:Quarter wbin:City)] ;
      owl:ObjectPropertyRange wbin:MonthlyWasteAmount ;
      rdfs:range wbin:MonthlyWasteAmount .

wbin:producedAmount rdf:type owl:ObjectProperty ;
      owl:ObjectPropertyDomain [a owl:Class; owl:ObjectUnionOf
(wbin:YearlyWasteAmount wbin:MonthlyWasteAmount)] ;
      rdfs:domain [a owl:Class; owl:unionOf (wbin:YearlyWasteAmount
wbin:MonthlyWasteAmount)] ;
      owl:ObjectPropertyRange wbin:WasteAmount ;
      rdfs:range wbin:WasteAmount .

wbin:type rdf:type owl:ObjectProperty ;
      owl:ObjectPropertyDomain wbin:WasteAmount;
      rdfs:domain wbin:WasteAmount;
      owl:ObjectPropertyRange wbin:Garbage ;
```

```
     rdfs:range  wbin:Garbage .

wbin:collects rdf:type owl:ObjectProperty ;
     owl:ObjectPropertDomain wbin:WasteBin;
     rdfs:domain wbin:WasteBin;
     owl:ObjectPropertyRange wbin:Garbage ;
     rdfs:range wbin:Garbage .

wbin:contributesTo rdf:type owl:Objectproperty ;
     owl:ObjectPropertDomain wbin:MonthlyWasteAmount;
     rdfs:domain wbin:MonthlyWasteAmount;
     owl:ObjectPropertyRange wbin:YearlyWasteAmount ;
     rdfs:range wbin:YearlyWasteAmount .



# data properties
wbin:year rdf:type owl:DataTypeProperty ;
     owl:DataTypePropertyDomain wbin:WasteAmount;
     owl:DataTypePropertyRange xsd:gYear .

wbin:monthOfYear rdf:type owl:DataTypeProperty ;
     owl:DataTypePropertyDomain wbin:MonthlyWasteAmount;
     owl:DataTypePropertyRange xsd:gYearMonth .

wbin:amount rdf:type owl:DataTypeProperty ;
     owl:DataTypePropertyDomain wbin:WasteAmount;
     owl:DataTypePropertyRange xsd:gYearMonth .


#  classes

# city
wbin:City rdf:type owl:Class ;
     rdfs:subClassOf geo:Feature ;
        owl:equivalentClass s:City , places:City .

# district
wbin:District rdf:type owl:Class ;
     rdfs:subClassOf geo:Feature ;
     geo:sfWithin wbin:City .

# quarter
wbin:Quarter rdf:type owl:Class ;
     rdfs:subClassOf s:Place , geo:Feature ;
     geo:sfWithin wbin:District ;
        geo:sfWithin wbin:City .

#------------------------------------------
#        WASTE BINS
#------------------------------------------

# waste bin
wbin:WasteBin rdf:type owl:Class ;
     rdfs:subClassOf geo:Feature ;
```

```
      geo:sfWithin wbin:District ;
      geo:sfWithin wbin:Quarter .

# Dry waste bin storing undifferentiated rubbish
wbin:DryWasteBin rdf:type owl:Class ;
      rdfs:subClassOf wbin:WasteBin ;
      rdfs:subClassOf [ a owl:Restriction; owl:onProperty wbin:collects;
owl:ObjectAllValuesFrom wbin:DryRubbish].

# Glass and Aluminum trash bin
wbin:GlassBin rdf:type owl:Class ;
      rdfs:subClassOf wbin:WasteBin ;
      rdfs:subClassOf [ a owl:Restriction; owl:onProperty wbin:collects;
owl:ObjectAllValuesFrom wbin:GlassOrAluminumRubbish].

# Organic trash bin
wbin:OrganicBin rdf:type owl:Class ;
      rdfs:subClassOf wbin:WasteBin ;
      rdfs:subClassOf [ a owl:Restriction; owl:onProperty wbin:collects;
owl:ObjectAllValuesFrom wbin:GlassOrAluminumRubbish].

# Paper trash bin
wbin:PaperBin rdf:type owl:Class ;
      rdfs:subClassOf wbin:WasteBin ;
      rdfs:subClassOf [ a owl:Restriction; owl:onProperty wbin:collects;
owl:ObjectAllValuesFrom wbin:OrganicRubbish].

# Plastic trash bin
wbin:PlasticBin rdf:type owl:Class ;
      rdfs:subClassOf wbin:WasteBin ;
      rdfs:subClassOf [ a owl:Restriction; owl:onProperty wbin:collects;
owl:ObjectAllValuesFrom wbin:PlasticRubbish].

# Used Clothes bin
wbin:UsedClothesBin rdf:type owl:Class ;
      rdfs:subClassOf wbin:WasteBin ;
      rdfs:subClassOf [ a owl:Restriction; owl:onProperty wbin:collects;
owl:ObjectAllValuesFrom wbin:UsedClothes].

#-----------------------------------------
#        WASTE / GARBAGE
#-----------------------------------------

# general Waste root,
wbin:Garbage rdf:type owl:Class .

# Plastic
wbin:PlasticRubbish rdf:type owl:Class ;
      rdfs:subClassOf wbin:Garbage .

# Glass or Aluminum
wbin:GlassOrAluminumRubbish rdf:type owl:Class ;
      rdfs:subClassOf wbin:Garbage .

# Paper
```

```
wbin:PaperRubbish rdf:type owl:Class ;
      rdfs:subClassOf wbin:Garbage .

# Dry rubbish
wbin:DryRubbish rdf:type owl:Class ;
      rdfs:subClassOf wbin:Garbage .

# Organic
wbin:OrganicRubbish rdf:type owl:Class ;
      rdfs:subClassOf wbin:Garbage .

# Used Clothes
wbin:UsedClothes rdf:type owl:Class ;
      rdfs:subClassOf wbin:Garbage .

#-----------------------------------------
#     PRODUCTION RATES
#-----------------------------------------

# waste amount
wbin:WasteAmount rdf:type owl:Class ;
      rdfs:subClassOf gr:QuantitativeValue ;
      rdfs:subClassOf muo:QualityValue .

#yearly waste amount
wbin:YearlyWasteMount rdf:type owl:Class .

#monthly waste amount
wbin:MonthlyWasteMount rdf:type owl:Class .
```

10. Annex V – Example of REST-API interactions

This chapter aims to provide complete examples of RESTful operations on sample resources. It refers to processing stages as introduced in section 3.3.1.

10.1 Resource creation

Default handlers for any request and resource type are preinstalled in LSM. They assume a simple, flat resource model where only the top-level predicates of a resource are considered. In the following an example `rdfa:PrefixMapping` instance will be used to demonstrate RESTful interactions and the involved handlers. The resource URI and predicates affected are highlighted:

```
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:rdfa="http://www.w3.org/ns/rdfa#"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema#" >
  <rdfa:PrefixMapping rdf:about="urn:res:prefixMappingTest">
    <rdfs:label xml:lang="en">Test namespace prefix</rdfs:label>
    <rdfa:prefix rdf:datatype="xsd:string">test</rdfa:prefix>
    <rdfs:seeAlso rdf:resource="http://prefix.cc/test"/>
    <rdfa:uri rdf:datatype="xsd:anyURI">http://schema.org/test#</rdfa:uri>
  </rdfa:PrefixMapping>
</rdf:RDF>
```

In order to "create" this resource the client passes its representation to the server, here by uploading the content of the file `prefixMappingTest.rdf` to the expected resource URI:

```
curl -H "Content-Type:application/rdf+xml" -X PUT --data-binary "@prefixMappingTest.rdf"
http://almanac.fit.fraunhofer.de:8082/api/resource/semantic/res:prefixMappingTest
```

Please note the usage of a compact URI to reference the resource within the request URL. The prefix `res` will be resolved against an existing `PrefixMapping` resource into the namespace '`urn:res:`'.

10.2 Resource validation

Initially the resource is de-serialized into a binary object model according to its mime type (indicated by the `Content-type` header. An error is thrown in case the supplied mime type is not supported or there are parsing errors. This implicit validation at representation level is optionally followed by a validation of the supplied resource content. There is no default resource validation in place, but a validation handler (SPARQL `ASK` query) for a particular resource or resource type may be uploaded as any other resource:

```
<rdf:RDF xmlns:ls="http://linksmart.eu/ontology#"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">
  <ls:AskQuery rdf:about="urn:res:ResourceValidatorPrefixMapping">
    <rdf:type rdf:resource="http://linksmart.eu/ontology#ResourceHandler"/>
    <rdfs:label>Validation query for prefix mapping resources</rdfs:label>
    <ls:service>
      <ls:ResourceValidator>
        <ls:targetClass rdf:resource="http://www.w3.org/ns/rdfa#PrefixMapping"/>
      </ls:ResourceValidator>
    </ls:service>
    <ls:source>
      <ls:ParameterizedSparqlString>
        <rdf:value rdf:datatype="http://www.w3.org/2001/XMLSchema#string"><![CDATA[
          PREFIX rdfa: <http://www.w3.org/ns/rdfa#>
          ASK
          {
           # Bin "prefix" and "uri" of local resource. Neither of both should be empty.
           {
            ?m a rdfa:PrefixMapping .
            ?m rdfa:prefix ?prefix. FILTER (strlen(str(?prefix)) != 0) .
            ?m rdfa:uri ?uri. FILTER (strlen(str(?uri)) != 0) .
           }
           # Neither "prefix" nor "uri" mapping should already exist.
           {
             SERVICE ?endpoint
              {
            FILTER NOT EXISTS { _:m1 rdfa:prefix ?p.  FILTER (str(?p) = str(?prefix)) } .
            FILTER  NOT EXISTS { _:m2 rdfa:uri ?u.  FILTER (str(?u) = str(?uri)) } .
              }
            }
          }
    ]]></rdf:value>
      </ls:ParameterizedSparqlString>
    </ls:source>
    <ls:input>
      <ls:Parameter ls:name="endpoint" rdfs:comment="SPARQL endpoint">
        <rdf:value rdf:resource="http://localhost:8888/Graphstore/query"/>
      </ls:Parameter>
    </ls:input>
    <ls:output>
      <ls:Result rdfs:comment="Boolean result of the validation">
        <rdf:value
          rdf:datatype="http://www.w3.org/2001/XMLSchema#boolean">false</rdf:value>
      </ls:Result>
    </ls:output>
  </ls:AskQuery>
</rdf:RDF>
```

The previous resource is an ordinary `ls:AskQuery` instance. LSM will treat it as resource handler by recognizing its type `ls:ResourceHandler` and retrieving the specification of provided to handle a particular target resource or resource type:

```
    <ls:service>
      <ls:ResourceValidator>
        <ls:targetClass rdf:resource="http://www.w3.org/ns/rdfa#PrefixMapping"/>
      </ls:ResourceValidator>
    </ls:service>
```

This `ResourceValidator` will be applied to resources of type `rdfa:PrefixMapping` and prevent that neither prefix nor namespace are empty or already in use. An attempt to supply an inadequate resource that violates that rules (i.e. not matches the validation query) will result in a 400 (Bad request) error.

10.3 Resource retrieval

Once created, the semantic resources are retrieved in the common RESTful way. A `GET` request is issued to the resource URI, optionally indicating the desired representation format (`Accept` header):

```
curl -H "Accept:application/rdf+xml"
http://almanac.fit.fraunhofer.de:8082/api/resource/semantic/res:prefixMappingTest

curl -H "Accept:application/rdf+json"
http://almanac.fit.fraunhofer.de:8082/api/resource/semantic/res:prefixMappingTest

curl -H "Accept:application/x-turtle"
http://almanac.fit.fraunhofer.de:8082/api/resource/semantic/res:prefixMappingTest

curl -H "Accept:text/n3"
http://almanac.fit.fraunhofer.de:8082/api/resource/semantic/res:prefixMappingTest
```

A `ReadRequestHandler` handler is used to excerpt the identified resource out of the graph continuum. The default handler is applicable for simple resources:

```
<rdf:RDF
    xmlns:ls="http://linksmart.eu/ontology#"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">
  <ls:ConstructQuery rdf:about="urn:res:ReadRequestHandlerResource">
   <rdf:type rdf:resource="http://linksmart.eu/ontology#ResourceHandler"/>
   <rdfs:label>Generic retrieval handler for flat resources</rdfs:label>
   <ls:service>
      <ls:ReadRequestHandler>
         <ls:targetClass rdf:resource="http://www.w3.org/2000/01/rdf-schema#Resource"/>
      </ls:ReadRequestHandler>
    </ls:service>
    <ls:source>
      <ls:ParameterizedSparqlString>
        <rdf:value rdf:datatype="http://www.w3.org/2001/XMLSchema#string"><![CDATA[
          CONSTRUCT { ?resource ?predicate ?object }
          WHERE {  ?resource ?predicate ?object }
        ]]></rdf:value>
      </ls:ParameterizedSparqlString>
    </ls:source>
    <ls:input>
      <ls:Parameter ls:name="resource" rdfs:comment="URI of resource to be retrieved" />
    </ls:input>
  </ls:ConstructQuery>
</rdf:RDF>
```

Clients may provide specific read handlers that recover more complex structures. Example of a `CONSTRUCT` query to retrieve a persistent Query/Update resource:

```
PREFIX  rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX  rdfa: <http://www.w3.org/ns/rdfa#>
PREFIX  ls:   <http://linksmart.eu/ontology#>
PREFIX  rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

CONSTRUCT {
    ?resource rdf:type ?type .
    ?resource rdfs:label ?label .
    ?resource rdfs:comment ?comment .
    ?resource ls:input ?input .
      ?input rdf:type ?input_type .
      ?input rdf:value ?input_value .
      ?input rdfs:label ?input_label .
      ?input rdfs:comment ?input_comment .
      ?input ls:name ?input_name .
      ?input ls:index ?input_index .
    ?resource ls:output ?output .
      ?output rdf:type ?output_type .
      ?output rdf:value ?output_value .
      ?output rdfs:label ?output_label .
      ?output rdfs:comment ?output_comment .
      ?output ls:name ?output_name .
    ?resource ls:source ?source .
      ?source rdf:type ?source_type .
      ?source rdf:value ?source_value .
}
WHERE
  {
    ?resource rdf:type ?type .
    ?resource ls:source ?source .
      ?source rdf:type ?source_type .
      ?source rdf:value ?source_value .
    OPTIONAL
      { ?resource rdfs:label ?label }
    OPTIONAL
```

```
                { ?resource rdfs:comment ?comment }
              OPTIONAL
                { ?resource ls:input ?input }
              OPTIONAL
                { ?input ls:name ?input_name }
              OPTIONAL
                { ?input ls:index ?input_index }
              OPTIONAL
                { ?input rdf:type ?input_type }
              OPTIONAL
                { ?input rdf:value ?input_value }
              OPTIONAL
                { ?input rdfs:label ?input_label }
              OPTIONAL
                { ?input rdfs:comment ?input_comment }
              OPTIONAL
                { ?resource ls:output ?output }
              OPTIONAL
                { ?output ls:name ?output_name }
              OPTIONAL
                { ?output ls:index ?output_index }
              OPTIONAL
                { ?output rdf:type ?output_type }
              OPTIONAL
                { ?output rdf:value ?output_value }
              OPTIONAL
                { ?output rdfs:label ?output_label }
              OPTIONAL
                { ?output rdfs:comment ?output_comment }
            }
```

10.4 Resource listing

A `GET` request to a factory resource (resource class) results in listing a short representation of all related resources (class instances):

```
curl -H "Accept:application/rdf+xml"
http://almanac.fit.fraunhofer.de:8082/api/resource/semantic/rdfa:PrefixMapping
```

The default `ListRequestHandler` will be used to retrieve the type and annotation properties:

```
<rdf:RDF xmlns:ls="http://linksmart.eu/ontology#"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">
  <ls:ConstructQuery rdf:about="urn:res:ListRequestHandlerResource">
    <rdf:type rdf:resource="http://linksmart.eu/ontology#ResourceHandler"/>
    <rdfs:label>Generic listing handler for flat resources</rdfs:label>
    <ls:service>
      <ls:ListRequestHandler>
        <ls:targetClass rdf:resource="http://www.w3.org/2000/01/rdf-schema#Resource"/>
      </ls:ListRequestHandler>
    </ls:service>
    <ls:source>
      <ls:ParameterizedSparqlString>
        <rdf:value rdf:datatype="http://www.w3.org/2001/XMLSchema#string"><![CDATA[
          PREFIX rdf:   <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
          PREFIX rdfs:  <http://www.w3.org/2000/01/rdf-schema#>
          CONSTRUCT {
            ?resource rdf:type ?type .
            ?resource rdf:value ?value .
            ?resource rdfs:label ?label .
            ?resource rdfs:comment ?comment .
          }
          WHERE {
            OPTIONAL { ?resource rdf:type ?type }
            OPTIONAL { ?resource rdf:value ?value }
            OPTIONAL { ?resource rdfs:label ?label }
            OPTIONAL { ?resource rdfs:comment ?comment }
          }
    ]]></rdf:value>
      </ls:ParameterizedSparqlString>
    </ls:source>
    <ls:input>
      <ls:Parameter ls:name="type" rdfs:comment="URI of resource type to be listed" />
    </ls:input>
  </ls:ConstructQuery>
</rdf:RDF>
```

10.5 Resource deletion

Equally to resource retrieval, the generic handler of deletion requests applies to simple resources only and custom handlers should be provided to handle more complex cases (e.g. removal of cross-references):

```
<rdf:RDF xmlns:ls="http://linksmart.eu/ontology#"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">
  <ls:DeleteUpdate rdf:about="urn:res:DeleteRequestHandlerResource">
    <rdf:type rdf:resource="http://linksmart.eu/ontology#ResourceHandler"/>
    <rdfs:label>Generic removal handler for flat resources</rdfs:label>
    <ls:service>
      <ls:DeleteRequestHandler>
         <ls:targetClass rdf:resource="http://www.w3.org/2000/01/rdf-schema#Resource"/>
      </ls:DeleteRequestHandler>
    </ls:service>
    <ls:source>
      <ls:ParameterizedSparqlString>
        <rdf:value rdf:datatype="http://www.w3.org/2001/XMLSchema#string"><![CDATA[
          DELETE { ?resource ?predicate ?object }
          WHERE {  ?resource ?predicate ?object }
    ]]></rdf:value>
      </ls:ParameterizedSparqlString>
    </ls:source>
    <ls:input>
      <ls:Parameter ls:name="resource" rdfs:comment="URI of resource to be deleted" />
    </ls:input>
  </ls:DeleteUpdate>
</rdf:RDF>
```

The default handler applies for this request and results in deletion of the previously created resource:

```
curl -X DELETE http://almanac.fit.fraunhofer.de:8082/api/resource/semantic/res:prefixMappingTest
```

11. Annex VI – Software prototype description

11.1 Source files

The source files of the Semantic Representation Layer (SRL) including the LinkSmart Metadata Framework (LSM) are part of the ALMANAC Git repository:

https://scm.fit.fraunhofer.de:8181/scm/git/almanac

The SRL files are located within the folder `platform/SemanticRepresentationLayer/`:

```
├── almanac-srl-res: SRL resource bundle (fragment bundle)
└── ontologies: ALMANAC specific ontologies and data schemas
```

The folder structure and naming beneath the path `platform/LinkSmart/` follows the LinkSmart repository[35] convention:

- Local copy of generic Maven LinkSmart artefacts required to build and deploy the software

```
├── linksmart-parent
├── linksmart-features
├── linksmart-osgi-component
```

- Separate folder grouping the interface bundles (to be merged in LinkSmart APIs in the future[36])

```
├── linksmart-apis
│   ├── linksmart-api-resource
│   ├── linksmart-api-metadata
```

- Individual implementation and resource bundles

```
├── linksmart-resource-impl: Default implementation of the Resource framework
├── linksmart-metadata-impl: Default implementation of the Metadata framework
├── linksmart-metadata-res: Metadata resource bundle (fragment bundle)
└── ontologies: LinkSmart specific ontologies and data schemas
```

Releases of ontology and data schemas will additionally be published at the locations:

- http://almanac.fit.fraunhofer.de/ontologies/SRL
- http://almanac.fit.fraunhofer.de/ontologies/LSM

11.2 Documentation

The generated JavaDoc documentation of is available at the locations:

- http://almanac.fit.fraunhofer.de/javadoc/SRL: SRL resource bundle documentation
- http://almanac.fit.fraunhofer.de/javadoc/LSM: LSM documentation

11.3 Development tool set

- Apache Maven[37]: Java build tool used across the ALAMANC and LinkSmart middleware projects

[35] https://linksmart.eu/redmine/projects/linksmart-opensource
[36] https://linksmart.eu/redmine/projects/linksmart-opensource/repository/revisions/master/show/components/linksmart-apis
[37] http://maven.apache.org/

- Bundle plugin[38]: Maven plugin responsible for building and configuring OSGi Bundles
- Apache Felix Maven SCR Plugin[39]: Maven plugin used to generate OSGi Service Component Runtime descriptions out of inline annotations within the Java code

11.4 Deployment/runtime tool set

- Apache Karaf[40]: Feature-rich OSGi runtime with an excellent community support
- Jersey[41]: Reference implementation of the JAX-RS standard used to deliver remote APIs
- OSGi - JAX-RS 2.0 Connector[42]: OSGi bundles used to publish OSGi service interfaces via Jersey
- Apache Jena[43]: Popular library and tool box for processing of Semantic Web data and reasoning

11.5 Remote service endpoints

Remote endpoints compliant with the specification given in this document have been installed at:

- REST API: http://almanac.fit.fraunhofer.de:8082/api/resource/semantic/
- RPC Query API: http://almanac.fit.fraunhofer.de:8082/api/service/sparql/query
- RPC Update API: http://almanac.fit.fraunhofer.de:8082/api/service/sparql/update

[38] http://felix.apache.org/site/apache-felix-maven-bundle-plugin-bnd.html
[39] http://felix.apache.org/documentation/subprojects/apache-felix-maven-scr-plugin.html
[40] http://karaf.apache.org/
[41] https://jersey.java.net/
[42] https://github.com/hstaudacher/osgi-jax-rs-connector
[43] https://jena.apache.org/